ReClick - A Modular Dataplane Design Framework for FPGA-Based Network Virtualization

Deepak Unnikrishnan, Justin Lu, Lixin Gao and Russell Tessier
Dept. of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
unnikrishnan@ecs.umass.edu

ABSTRACT

Network virtualization has emerged as a powerful technique to deploy novel services and experimental protocols over shared network infrastructures. Although recent research has highlighted field programmable gate arrays (FPGAs) as attractive platforms for high performance network virtualization, these devices remain inaccessible to the larger networking research community due to the absence of user-friendly programming models. A programming model that can abstract the intricacies of the hardware platform while being aware of the underlying resource constraints is highly desirable. In this paper, we present ReClick, a framework to efficiently design and deploy reconfigurable dataplanes for FPGA-based network virtualization systems. A hardware-agnostic programming model is described that allows developers to focus on the virtual dataplane semantics rather than the implementation details. The framework exposes interfaces similar to the popular software router development framework, Click, and promotes design reuse. Optimization strategies are included in ReClick which use similarities between virtual dataplane configurations to implement multiple planes in an area-efficient manner. Dataplanes exhibiting up to 1 Gbps data rate have been automatically compiled and tested in hardware in a NetFPGA platform.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Experimentation, Performance

Keywords

Network Virtualization, FPGA, Programming models, Click modular router, NetFPGA

1. INTRODUCTION

Over the past decade, the Internet has significantly evolved to support new applications and services. As the demand for application performance continues to grow, new protocols and architectures will need to be deployed in the future Internet. Network service providers are often wary of the economic consequences of deploying experimental protocols and architectures on their stable network infrastructures. The construction of separate physical networks to meet these requirements is generally economically unviable. In many cases, a common physical substrate that minimizes operating costs, investment and power consumption is desirable. Network virtualization provides a powerful solution to this problem by allowing multiple virtual networks to operate over shared network resources [7] [14] [24]. Such an approach allows new protocols and experimental services to be deployed on legacy network infrastructures.

A physical network can be virtualized via routing resource sharing across multiple distinct *virtual* routers. Often, each virtual network has unique requirements regarding the way packets are processed. The packet processing features of a virtual network are implemented in its *dataplane*. The experimental nature of protocols implemented in virtual networks require a highly customizable dataplane to guarantee flexibility. Additionally, virtual networks demand a high degree of isolation and performance. Conventional approaches for network virtualization use software techniques such as full or para virtualization where software resources such as CPU cycles and the physical memory of the physical router are shared between the virtual routers [8]. Although software-based techniques offer considerable flexibility, the performance of virtual networks is limited by the sequential nature of execution in microprocessors.

Recent research has identified field programmable gate arrays (FPGAs) as attractive platforms for network virtualization by virtue of their adaptability, specialization and fine-grained parallelism [5] [18] [25]. FPGA-based virtualization platforms have demonstrated up to two orders of magnitude better performance than their software counterparts in terms of packet throughput. However, the widespread adoption of FPGA-based network virtualization systems requires access to high-level programming models that allow a user to express complex dataplane features without compromising the performance and area efficiency of the implemented hardware design. Although existing hardware description languages, such as Verilog and VHDL, can gener-

ate area-efficient and high-performance hardware for complex dataplane functions, they expose an unfamiliar programming interface to many network developers. Recently, several frameworks that expose abstract interfaces have been proposed [9] [22] to address this issue. However, these approaches can be limited in their ability to use the logic and memory resources of a hardware platform in an area-efficient manner.

In this paper, we present ReClick, a framework to design and deploy custom dataplanes for FPGA-based network virtualization that are modular and area-efficient. ReClick abstracts the intricacies of reconfigurable hardware design by providing the dataplane designer a language and compiler which are sufficient to express common packet processing operations. The framework exposes an interface which is similar to Click [2], the widely-used dataplane design framework for software virtual routers. The framework features optimizations to maximize packet forwarding performance and resource efficiency in reconfigurable hardware for multiple dataplanes. Designs are automatically compiled to FPGA hardware without extensive user intervention. A validation flow based on register transfer level (RTL) simulation is also in place for debugging and assessment prior to hardware deployment. A collection of pluggable modules which can be used with the framework have been developed and made available to the research community. The effectiveness of the framework is demonstrated with two dataplane design examples - an IPv4 router and an IP router enhanced with onion routing capabilities. These dataplanes have been verified on a Virtex II FPGA available on the NetFPGA platform.

The rest of the paper is organized as follows: Section 2 introduces FPGA-based network virtualization and presents related work on programming models for FPGA-based packet processing. Section 3 describes the modular network virtualization platform and introduces the ReClick programming model. Section 4 presents the FPGA-based platform used for experimentation and evaluates the performance of the framework. The paper is summarized in Section 5 and directions for future work are offered.

2. BACKGROUND

2.1 Network Virtualization

A virtual network represents a programmable *slice* of the physical network providing fixed bandwidth and QoS guarantees. Flexibility is a key requirement of any virtual networking substrate. Specifically, a virtual network must offer maximum control over its dataplane to implement custom packet forwarding functions. For example, the deployment of new addressing schemes such as ROFL [16] requires customization of nearly all aspects of the network core such as the routing protocol and the address lookup algorithm. Other examples include QoS schemes that require security mechanisms such as network anonymity or onion routing [13] [28]. The experimental nature of virtual networks, however, demands that dataplane flexibility does not come at the expense of long design cycles. An ideal virtualization platform must also scale to support a large number of programmable slices without considerable degradation of packet forwarding performance.

Conventional network virtualization techniques use off-the-shelf hardware and host virtualization techniques to share router resources among virtual networks [7] [8]. A comprehensive survey of software-based virtualization techniques can be found in [11]. Software approaches offer considerable design flexibility and ease of use, although the execution of virtual routers in operating system and higher layers severely limit their packet forwarding performance. For example, container-based virtualization techniques such as OpenVZ [4] can support only up to 300 Mbps with 64-byte packets [7].

The continued demand for high performance in virtualized networks has motivated hardware-based virtualization strategies. Several commercial custom ASIC-based platforms have been recently proposed [1] [24]. These approaches, however, do not offer a level of design flexibility demanded by experimental virtualization platforms. FPGAs offer a nice design tradeoff between these two approaches by virtue of their reconfiguration properties and availability of abundant parallelism. Recent FPGA-based virtualization systems [5] [27] have demonstrated up to two orders of magnitude faster throughput than previous software-based approaches.

2.2 Programming Models for FPGA-based Packet Processing Systems

Programmability is a key issue for FPGA-based packet processing systems. Although many of these systems [3] use RTL description languages like Verilog and VHDL, the hardware expertise required to master these languages makes them inaccessible to the larger community of network developers. Several recent research attempts try to address this issue by providing high-level programming descriptions that abstract the details of the underlying hardware. Horta et al. [15] provide a first attempt to introduce programmability in FPGA-based packet processing systems. A module-based approach to implement reconfigurable high speed packet processing circuits is presented. Dynamic hardware plugins are assembled in hardware for single data planes using a restrictive set of directives. In contrast, our approach provides a flexible high-level interface to the user and support for multiple virtual dataplanes.

NetThreads [17] uses multiprocessors constructed from the FPGA fabric (soft multiprocessors) to implement packet processing features. The soft microprocessors are embedded within the packet processing data path of a NetFPGA card. Packet processing features are described using C programs that execute on the multiprocessor system. A modified GCC toolset is used to generate executable binaries for the soft microprocessors. Writing C-style programs greatly simplifies the task of the application designer. However, the multiple cycles required to execute packet processing tasks limit the packet forwarding performance of this approach to 5,000 packets per second.

Click [2] is a widely popular framework for building software routers. Click allows users to write *configurations* that describe packet processing functions as a graph of interconnected modules called *elements*. While configurations are written in a custom Click language, the behavior of individual elements can be described in C++. The elements are interconnected through *ports* that either actively forward (*push*) or passively receive (*pull*) data. Click has been

Table 1: Feature comparison of programming models for FPGA-based packet processing systems

Framework	Frontend	Virtualization support	Module selection
[10]	XML	No	Static
NetThreads	C	No	NA
G	G, Click	No	Static
Chimpp	Verilog HDL, Click	No	Static
SwitchBlade	Verilog HDL	Yes	Dynamic
ReClick	ReClick, Verilog HDL, Click	Yes	Dynamic

widely adopted in network research by virtue of it's simple design and the availability of a diverse collection of reusable open source modules.

Nikander et al. [21] propose a tool chain that compiles C++-based Click elements to synthesizable Verilog descriptions. In this approach, Click elements described in C++ are first transformed into an intermediate representation (LLVM). The LLVM is a collection of modular components for building compiler tool chains which includes a number of code optimizers and backends for hardware. The optimized code is converted back into C code. The C program can be taken through 3rd party C-to-Verilog synthesis tools such as AHIR [23] to generate hardware descriptions. Although this approach enables existing Click descriptions to be easily migrated to reconfigurable hardware, it has many practical limitations. Click, for instance, uses certain features in C++ such as virtual functions and polymorphism, that are difficult to directly implement in hardware. Furthermore, the packet forwarding performance of generated configurations have not been reported.

Brebner et al. [10] propose a system that can compile finite state machines described using high level XML descriptions to FPGA bitstreams. The packet processing system is composed of *threads* and *hooks*. Threads represent a unit of concurrency in the programmable logic while *hooks* provide wrappers around unconventional packet processing blocks to be interfaced to the system. The programming model, however, constrains designers to use finite state machine models, a rather nonintuitive way to describe packet processing blocks.

The G [9] [20] framework represents a first attempt to convert packet processing descriptions in a high-level language to synthesizable Verilog descriptions. G uses a design philosophy that is similar to the one used by Click. Packet processing is specified as a pipeline of interconnected modules. A module can perform simple operations on the packet such as "set a field in the packet", "insert a field after an offset in the packet" or "push a packet through a specific port". The G language infrastructure includes a simulator and debugger for functional verification of designs. Complex packet processing operations such as packet switching and scheduling are not yet supported. Additionally, the proprietary nature of the framework, the lack of availability of a library of modules and the use of Xilinx-specific interconnect technology are likely to affect the popularity of the framework.

Chimpp [22] is a framework similar to G for writing Click-style packet processing descriptions on the NetFPGA platform. Modules can be parameterized using XML descriptions. Unlike G, Chimpp allows configurations to be composed of a combination of hardware and software elements. However, the behavior of hardware-specific elements must be described using Verilog/VHDL, limiting access to typical network programmers.

SwitchBlade [6] takes an alternative approach by providing a model that allows packet processing modules to be swapped in and out of the reconfigurable hardware without the need to resynthesize the hardware. Frequently-used hardware blocks are presynthesized to the FPGA in advance. Users select a subset of modules that are required to process the packet through register interfaces. The selection is later encoded in a bitmap header which is appended to incoming packets. Each module in the datapath examines the bitmap and decides whether or not to process the packet. Presynthesized elements as well as new modules need to be written in Verilog which may be a challenge for networking researchers who are not familiar with hardware design.

Table 1 summarizes the features supported in previously discussed frameworks. In general, these efforts are either proprietary or require designers to be familiar with hardware design knowledge. Except SwitchBlade, none of the frameworks provide a straightforward approach to virtualize the hardware. ReClick addresses these issues by offering a flexible and open platform for virtual dataplane design using reconfigurable hardware. A modular design environment is used with a Click-style frontend that allows existing Click configurations to be migrated to reconfigurable hardware with minimal changes. New modules, designed in a hardware-agnostic language, can be dynamically reused between multiple dataplanes. The generated designs can be easily deployed on open hardware platforms like NetFPGA.

3. DESIGN

Our paper makes the following specific contributions:

1. An architecture for FPGA-based network virtualization featuring extensible modular dataplane components. The system supports component reuse between multiple active virtual dataplanes in the FPGA. Pipelining is used within components to achieve the highest packet forwarding rates. The operations on packets are scheduled to minimize packet forwarding latency.
2. A software framework that describes common packet processing features of virtual dataplanes as a permutation of simple operations on packets, hiding hardware implementation details. A compilation framework that

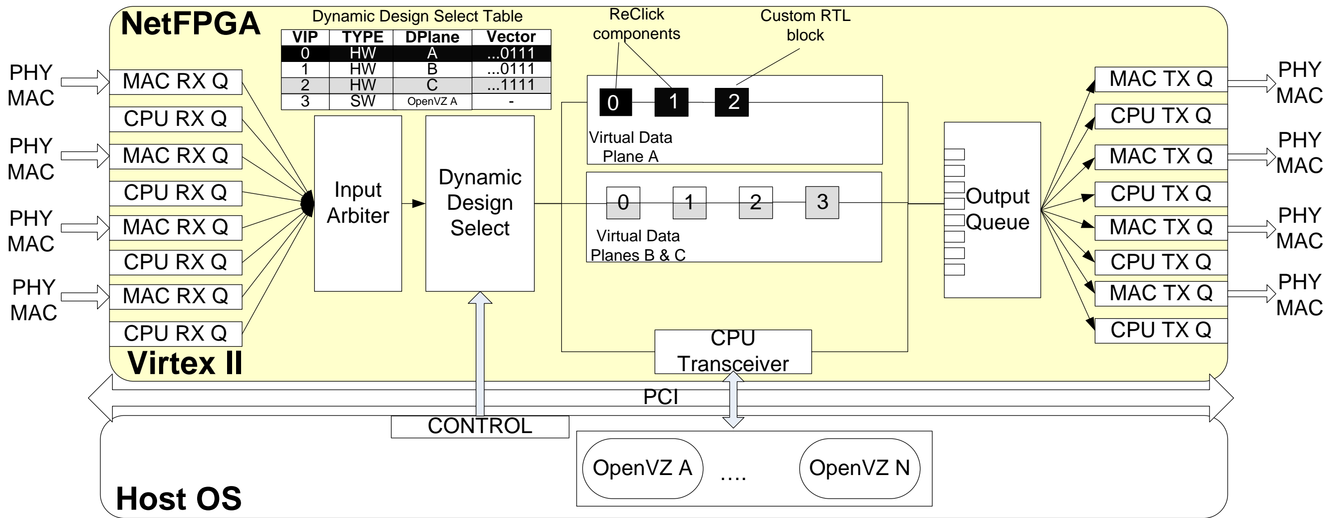

Figure 1: Overview of the FPGA-based virtualization platform

can translate these descriptions to area-efficient hardware descriptions.

3. A Click-like interface to compose and deploy virtual dataplanes from reusable dataplane components.

3.1 Architecture of the Virtualization Platform

Our ReClick system is explained in the context of an existing FPGA-based network virtualization platform. The NetFPGA-based system [25] supports heterogeneous virtual dataplanes implemented in both FPGA and host software. High-throughput virtual dataplanes are synthesized and configured in a Virtex II FPGA, while multiple low-throughput virtual dataplanes are implemented in OpenVZ containers in the host operating system. The packet processing datapath consists of an input arbiter, a dynamic design select module, several output port lookup modules and an output queue module.

Packets arriving at physical Ethernet interfaces are polled by the input arbiter. The dynamic design select module classifies the packets to one of the several virtual dataplanes implemented in the FPGA or in the host software. The hardware dataplanes are implemented by replicating output port lookup modules [26] of the NetFPGA reference router [3]. Each hardware dataplane includes custom forwarding logic and forwarding tables to process packets. Processed packets are dispatched through one of the several physical Ethernet interfaces available on the NetFPGA card. The CPU transceiver module within the FPGA is used to transmit and receive packets from OpenVZ-based virtual dataplanes in host software.

Figure 1 shows the architecture of our network virtualization platform used with ReClick. The architecture implements two specific extensions to support extensible and modular virtual dataplanes. First, the forwarding logic resources previously implemented using output port lookup modules [25] are organized as a hierarchical pipeline of smaller packet processing units. Each unit represents an independent packet processing entity with several streaming interfaces. The framework facilitates the integration of two types of packet processing units namely *ReClick components* and *custom RTL blocks* (see Figure 1). The fundamental difference between these two types of units lies in the way they describe packet processing behavior. ReClick components (hereafter referred to as *components*) are specified in the domain specific language discussed in Section 3.2 as a permutation of simple packet processing primitives.

The decomposition of virtual dataplanes into independent packet processing units provides opportunities for design reuse within the shared network virtualization platform. Consider, for example, a virtual dataplane that describes a new protocol, such as path splicing [19]. Such a dataplane performs several conventional IP processing tasks such as time-to-live (TTL) and checksum updates. In many cases, the similarity between the virtual dataplanes can be exploited to reduce the area overhead of implementing virtual dataplane features separately in the FPGA-based network virtualization platform. For example, a new virtual dataplane can be deployed by adding a few components to an existing virtual dataplane configuration or by reusing a subset of the existing dataplane components.

To facilitate resource sharing, the dynamic design select table (in Figure 1) has been modified to associate a 32-bit *bitvector tag* (Vector in Figure 1) with each incoming packet. The bitvector tag, programmed from software through a user register, is used to select those virtual dataplane components that are required to process the packet. Each bit in the bitvector corresponds to a component in the virtual dataplane. For simplicity, we reserve the lower order bits in the bitvector for those components of the virtual dataplane that process incoming packets first. A bit corresponding to a component is set if that particular component is used to process the packet. Each component in the virtual dataplane checks its bit position in the bitvector tag associated with the packet. If the bit is set for the incoming packet, it is processed by the component. Otherwise, the packet is

Table 2: Reclick Primitives

Primitive	Arguments	Description	Conditional Execution?
get	{field, packet}	Extracts the field from the packet word. Assigns to user variable	No
set	{field, packet, value}	Set field of packet to desired value-variable	Yes
insert	{value, position, packet}	Insert a user defined field at the given position in the packet	No
remove	{field, packet}	Removes a user defined field from the packet	No
assign	{packet, port}	Assign inputs to packets or packets to output ports	Yes

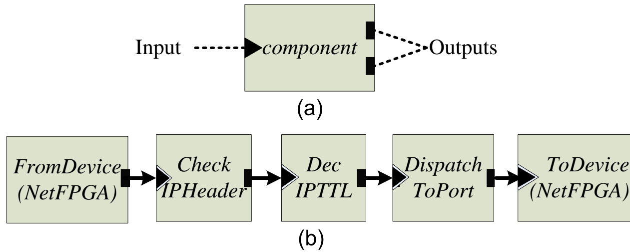

Figure 2: (a) A basic component and (b) Use in a virtual dataplane configuration

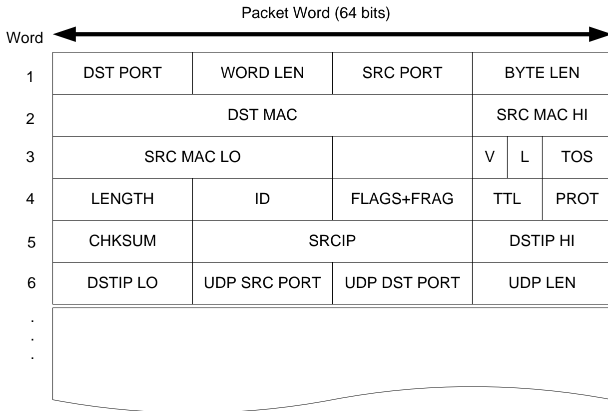

Figure 3: An IPv4 packet word processed by NetFPGA reference router (from [3])

simply forwarded to the next module.

As an example, consider three virtual networks - black, white and grey as shown in Figure 1. The black virtual network does not share components with any other virtual network and hence, has its own dedicated routing resources. The white and gray virtual networks, however, share routing components (except component 3). In this case, a single dataplane configuration (C), is sufficient to address the requirements of both the virtual networks. The bit vector configuration for all the networks are indicated in Figure 1.

3.2 ReClick Programming Model

Our framework exposes two types of programming interfaces to application developers. The first interface facilitates the development of independent packet processing components

Program 1 Click description of the virtual dataplane configuration in Figure 2

```
//Instantiate components
src::FromDevice(NetFPGA);
checkip::CheckIPHeader();
ttl::DecIPTTL();
dispatch::DispatchToPort();
sink::ToDevice(NetFPGA);

//Interconnect component instances
src[out]->[in]checkip[out] ->[in]ttl[out]->
[in]dispatch[out]->[in]sink;
```

by combining a set of simple primitives. The second interface, which is similar to the software router development framework, Click, allows virtual dataplanes to be composed by stitching together multiple components.

Figure 2(a) shows a ReClick component. The component interfaces include a set of input/output ports which may include optional configuration parameters. The input ports of each component are actively driven by packet outputs from previous components. ReClick implements this *push* style dataflow in a manner similar to the Click modular router framework [2]. Several such components may be interconnected to form realistic virtual dataplane configurations. For example, Figure 2(b) shows a simple virtual dataplane configuration that accepts packets from the NetFPGA pipeline (e.g. from dynamic design selection in Figure 1) via the *FromDevice(NetFPGA)* component, filters non-IP packets (*CheckIPHeader*), decrements the TTL field in the packet (*DecIPTTL*), modifies the packet header to be forwarded through a specific NetFPGA physical interface (*DispatchToPort*) and forwards the packets to the rest of the NetFPGA pipeline (e.g. output queue) via the *ToDevice(NetFPGA)* component. Configurations can be formulated using Click style descriptions. An example of the Click formulation of the virtual dataplane in Figure 2(b) is shown in Program 1.

The behavior of individual components can be described in the domain specific language, ReClick, or, if preferred, by the dataplane designer, using conventional RTL descriptions. Like other domain specific languages [9], the packet is the central operational entity in a ReClick component. Packets vary in size and packet sizes can exceed the datapath width of the hardware pipeline.

Packet operations are therefore conducted as a sequence of operations on packet *words*. The packet word represents the largest quantum of packet data that can be accommodated using the hardware datapath in a single clock cycle. In a fully pipelined design, each packet word can be operated upon in a single clock cycle. Figure 3 shows the first few words of an IPv4 packet processed by the NetFPGA reference router [3]. The NetFPGA reference router uses a 64-bit wide datapath. The packet word consists of one or more *fields*, whose contents represent meaningful information. For example, the most significant 48 bits of word 2 indicates the destination MAC address, while the lower order bits 8 to 15 of word 4 indicate the TTL information. ReClick provides a set of primitives that can characterize frequent packet processing operations (Table 2). These primitives can be combined with our software infrastructure to form a virtual dataplane.

We illustrate the capabilities of ReClick by considering a simple design example *DecIPTTL*. DecIPTTL is a frequently-used packet processing component which is used to filter packets whose TTL values have expired (indicated by a value of zero in the TTL field). Program 2 describes the operation of a DecIPTTL component using the set of primitives presented in Table 2. The component interfaces include an input port (in0) and two output ports (out0, out1). Valid packets are forwarded via out0 to the next component while expired packets are dropped via out1. ReClick features two special datatypes - Packet and Field, in addition to standard datatypes. The Packet type is used to describe a packet, which is operated upon by the component as it transits from inputs to outputs. The Field type is used to define packet fields within words. ReClick represents a field as a tuple of two parameters - the index of the word relative to the start of the packet and the subset of meaningful bits within that word.

Standard data type variable declarations are associated with integer values that characterize the storage width. These values provide useful information for the ReClick compiler while inferring hardware components. All primitives, except **assign**, operate on packet words. The **get** and **set** primitives are used to modify packet field information. They are described in more detail in Section 3.3. Standard expressions can be used to modify variable data or field information. The insert and remove primitives (not shown in the example) allow custom user fields to be inserted or removed from specific bit positions within the packet word. Assign statements are used to associate packets arriving at the input ports of the component with packet variables.

If-else style conditional statements are supported for a subset of primitives as indicated in Table 2. Conditional statements enhance the expressiveness of the packet processing descriptions by adding flexibility to operate on packets based on static (compile-time) or run-time decisions. For example, wrapping set statements within conditional statements enables packet values to be conditionally modified. Similarly, conditional execution of assign statements allows packets to be scheduled across multiple ports. ReClick does not support straightforward implementation of conditionals for insert and remove statements to reduce hardware complexity.

Program 2 ReClick description of a DecIPPTL component

```
component DecIPTTL {

//I/O port declaration
input in0;
output out0;
output out1;
packet pkt;

//Define Time-to-live(TTL) field
field TTL [15:8] of word 4;

//A 32 bit integer to store TTL value
int ttl_val:32;
//Variable to store the new TTL
int ttl_val_dec:32;

//Packet behavior
assign in0 to pkt;

ttl_val = get TTL of pkt;
ttl_val_dec = ttl_val - 1;

//Conditionally set fields
if(ttl_val>0) {
   set TTL of pkt to ttl_val_dec;
} else {
   set TTL of pkt to ttl;
}

//Schedule packets to outputs
if(ttl_val>0) {
   assign pkt to out0;
} else {
   assign pkt to out1;
}
}
```

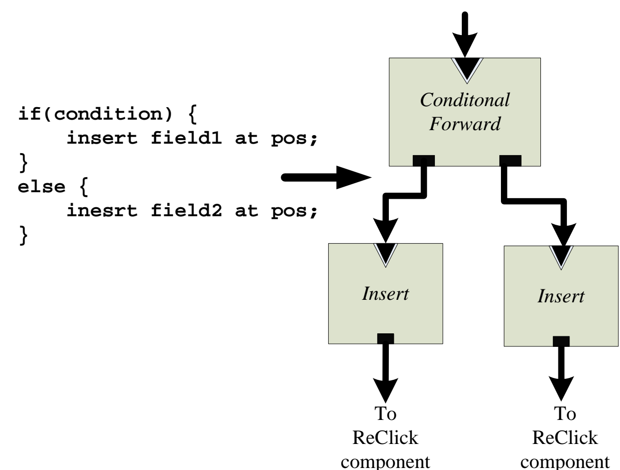

Figure 4: Conditional inserts/removals can be implemented in an indirect fashion using Click configurations. In this example, a conditional insertion is implemented as two separate ReClick components

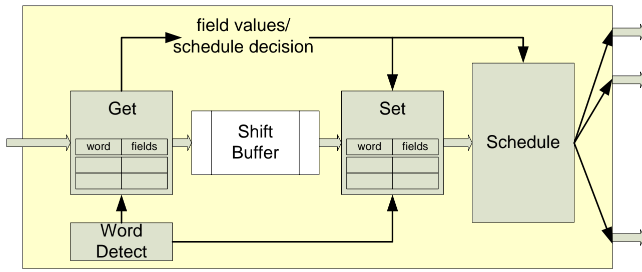

Figure 5: The generic architecture of a ReClick component

However, the programming model supports conditional inserts and removals in an indirect fashion.

Consider a scenario as shown in Figure 4 where two distinct fields need to be inserted at a specific position in the packet based on the falsity or trueness of a user-defined expression. The semantics of this feature can be correctly implemented with two ReClick components as shown in Figure 4. The *conditional forward* component checks the user-defined condition and pushes the packet through one of the two available ports. The ports are attached to two distinct insert modules that perform the insert operation.

ReClick allows special variables called handlers to be defined. The handler variables are modeled as simple memory elements that store configuration parameters or packet flow statistics within components. For example, a handler variable whose value is incremented on the receipt of a first packet word can be used to keep track of the number of packets handled by the particular component. ReClick models handler variables as user registers in hardware.

3.3 Hardware Model

Packet forwarding performance is critical to FPGA-based virtual dataplanes. As a result, a ReClick component is modeled as a hardware pipeline as shown in Figure 5. The ReClick compiler generates the elements of the pipeline according to the packet processing behavior specified by the user. Not all pipeline elements shown in the figure are required by all component descriptions. The pipeline consists of a collection of the following set of modules:

1. **get** - The *get* module implements a table that stores the words and fields of interest in the packet. Each incoming packet word is checked against this table to extract fields of interest. The contents of the table are sequenced by the ReClick compiler.
2. **set** - The *set* module is similar to *get* except that it is used for packet modification operations. The set module includes a table that stores fields and words that need to be modified. The module identifies fields of interest in the packet word and modifies them as they are clocked out of the component. The contents of the *set* table are sequenced by the ReClick compiler.
3. **insert** - The *insert* module inserts fields at specific positions within the packet word and adjusts the packet length. Additional words are inserted whenever necessary.
4. **remove** - The *remove* module removes fields of interest from specific positions in the packet word and adjusts the packet length.
5. **schedule** - The *schedule* module is responsible for inter-component flow control. Additionally, it provides the ability to conditionally forward packets between multiple ports.

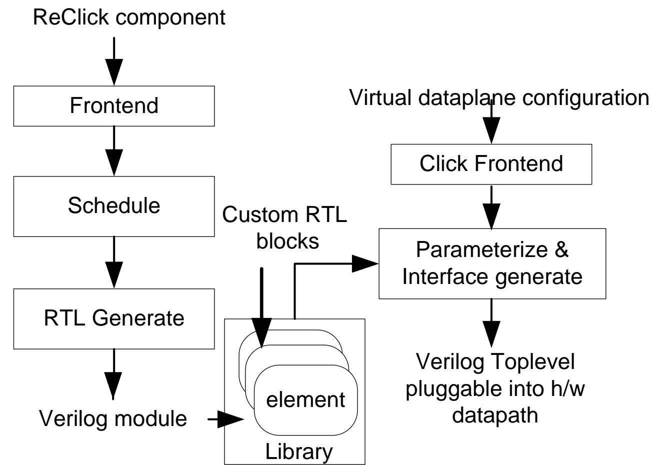

Figure 6: Compiler Framework

Packet forwarding at high throughput requires that each component is free from pipeline stalls. However, this condition is seldom the case. A write operation on a packet word whose value depends on information from words that are yet to be received by the pipeline causes a pipeline to stall. For example, a set operation on the DSTPORT (destination port) of word 1 in Figure 3 depends on the DSTIPHI field from word 5 and the DSTIPLO field from word 6 (destination IP address). This dependency causes the pipeline to stall at least for 6 cycles.

To address such *write after read* hazards, we introduce a shift buffer between the input and output ports. The size of the shift buffer is statically computed at compile time as the index of the farthest word from the first word of the packet, whose field values affect packet modification or scheduling decisions. For example, in the previous example, a shift register of 6 words is used. When packets arrive at the component's input ports, they are successively shifted through the shift module during every cycle. The shift buffer ensures that field information from all dependent words is available before packet modification or scheduling decisions are performed.

3.4 Design Flow

The phases of the ReClick framework are illustrated in Figure 6. ReClick behavioral descriptions are parsed and typechecked for errors by the frontend. The scheduler examines the description to detect operations on fields that can be scheduled in the same cycle. Specifically, fields belonging

to the same word can be scheduled in the same cycle. A wider hardware datapath allows longer packet words, and hence, more field operations to be sequenced in the same cycle. However, this advantage comes at the expense of a higher hardware cost. In general, the hardware datapath width represents an important area-tradeoff parameter for the virtual dataplane designer. For simplicity, we choose a 64-bit wide datapath which is similar to that used in the NetFPGA reference router architecture.

Operations that are dependent on field values from multiple packet words are scheduled according to the *as soon as possible (ASAP) schedule*. Such operations are immediately scheduled when all dependent information is available from the hardware pipeline. The backend uses the schedule information to generate register transfer level descriptions in Verilog HDL. Except for the shift buffer, all component features are generated on an as needed basis. The backend generates table entries for **get** and **set** modules within the component pipeline according to the schedule determined in the previous step. Parameterizable insert and remove modules are instantiated according to the component description. Finally, the compiler generates hardware structures, such as wires and registers, to stitch together the component pipeline.

To supplement user-defined components, automatically generated RTL descriptions are added to a library for use in subsequent designs. The library supports the inclusion of additional custom RTL blocks wrapped in standard streaming interfaces that conform to the NetFPGA reference datapath. The ReClick compiler generates an RTL description for each component. We have developed a collection of library components as shown in Table 3. Multiple such components can be instantiated using the ReClick frontend to produce a virtual dataplane description which is readily pluggable into the NetFPGA datapath.

3.5 Example ReClick Configurations

We illustrate two design examples to demonstrate the capabilities of ReClick.

3.5.1 IPv4 Router

Figure 7 illustrates an IPv4 router example designed from simple ReClick components. The first two modules (CheckIPHeader and DropBroadcast) are used to filter out non-IP and broadcast packets. The lookup module is a custom RTL block which is described in Verilog HDL. The module is available for designers from a library. The lookup module extracts the destination virtual IP address in the packet and looks it up in a ternary CAM-based forwarding table within the FPGA. It also features an ARP table to obtain the next-hop MAC information. The DecIPTTL module recalculates the time to live (TTL) values and filters out expired packets. Register interfaces for writing forwarding table entries and reading bookkeeping information are automatically inserted by the compiler. All components except Lookup are ReClick components. Lookup is a custom RTL module.

3.5.2 Onion router

Onion routing is a widely popular technique to implement secure and anonymous communication over public networks.

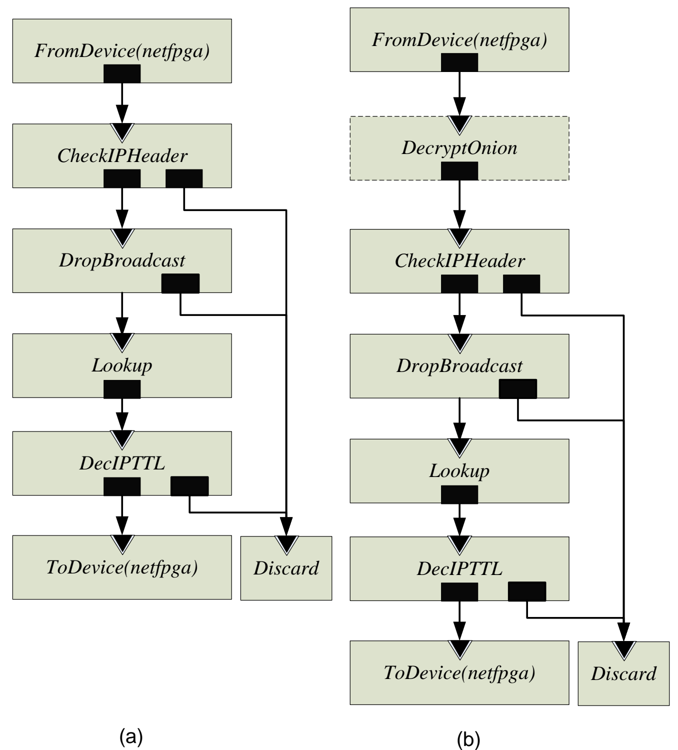

Figure 7: An IPv4 router. Subfigure (a) represents a standard router. Subfigure (b) includes onion router capabilities

The sender node chooses a set of onion routers to anonymously route a packet to the destination node. A path is constructed from this node set. The sender then wraps the packet using successive layers of encryption to create an *onion packet*. The onion is passed to successive onion routers, each of which removes a layer of encryption before forwarding the packet to the next intermediate router. The destination node removes the final layer of encryption to recover the packet data.

We implement an onion router in ReClick by extending the IPv4 router presented in the previous subsection. A decryption component is attached to the front of the data processing pipeline. While real onion routers use public-key cryptography to encrypt packets, we use a symmetric decryption algorithm for simplicity. The onion router shares all components except *DecryptOnion* with the standard IPv4 router. A single configuration, as illustrated in Figure 7(b), can be used for both dataplanes.

4. EVALUATION

We evaluate ReClick by comparing the packet forwarding performance and resource consumption of an IPv4 dataplane which is automatically generated by our framework against a hand-coded IPv4 reference router implementation which is available from the NetFPGA project. Additionally, we compare the ReClick IPv4 dataplane with equivalent dataplanes generated using Chimpp [22] and Switchblade [6] frameworks using similar metrics.

Table 3: Resource Utilization and Latency of ReClick components on Virtex II Pro

Element	Description	Slices	FFs	LUTs	Lines of Code	Latency (Cycles)
CheckIPHeader	Checks IP header and drops non-IP packets	192	324	160	223	5
DecIPTTL	Decrements TTL and drops expired packets	30	210	339	227	3
DecryptOnion	Decrypt packet data	1037	676	1155	291	6
Discard	Discard the packet	12	0	3	165	1
DispatchToPort	Forward packet through a specific port	666	324	167	180	1
DropBroadcast	Filter broadcast packets out	196	324	312	217	2
EtherMirror	Swap ethernet source and destination addresses	388	356	329	197	3
FromDevice	Interface to NetFPGA input datapath	0	0	0	53	0
IPMirror	Swap destination and source IP addresses	427	388	298	197	6
ToDevice	Interface to NetFPGA output datapath	0	0	0	54	0

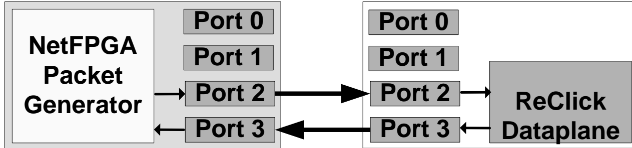

Figure 8: Topology for experiments using a packet generator and a dataplane

4.1 Packet forwarding performance

For performance evaluation, we compare the throughput of a single IPv4 virtual dataplane generated from ReClick against the NetFPGA reference router. The Virtex II FPGA can accommodate up to four IPv4 virtual dataplanes. Each virtual dataplane operates at a clock frequency of 62.5 MHz. Figure 8 shows the experimental setup for measuring packet throughput. The NetFPGA packet generator [12] is used to accurately generate traffic at line rate (1 Gbps). Packets of sizes varying from 64 bytes to 1024 bytes are used to flood the physical Ethernet interfaces of the target NetFPGA card.

Figure 9 compares the throughput of the ReClick modular router against the throughput of the NetFPGA reference router for varying workloads. The ReClick IPv4 router consistently handles line rate traffic for all packet sizes (1 Gbps) demonstrating that modular organization of the virtual dataplane does not impose any forwarding performance loss on the network virtualization platform. However, the individual components do introduce additional latency into the packet forwarding pipeline. These latencies are characterized in Table 3. The shift buffers between input and output ports prevent the increased latency from affecting packet throughput.

4.2 Resource consumption

Table 4 presents the logic resources consumed by the ReClick IPv4 router, an extended ReClick IPv4 router that supports packet encryption for onion routing and the NetFPGA reference router implementation. The resource utilization statistics were derived from Xilinx ISE 10.1 synthesis reports generated after the logic map step of the compilation process. All designs were subsequently mapped to silicon through ISE physical design (e.g. place, route, and bitstream generation). The ReClick IPv4 router consumes approximately 49.7% of

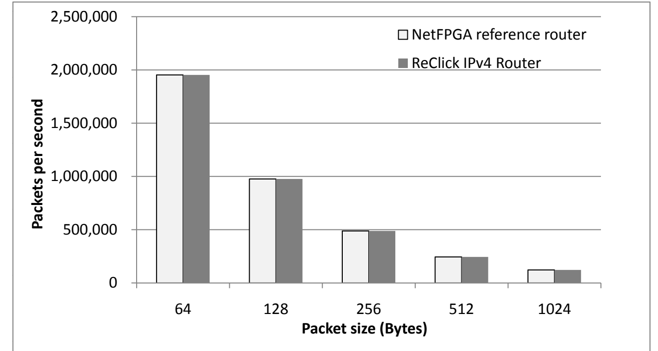

Figure 9: Packet forwarding throughput of ReClick IPv4 router and NetFPGA reference IPv4 router for varying packet sizes

the available 4 input lookup tables (LUTs) and 62% of the available slices. The logic utilization is thus comparable to that of a hand-coded reference design available from the NetFPGA development platform. However, the presence of shift buffers, which are realized using block RAM memories (BRAMs) and registers within the FPGA, increase the utilization of BRAM resources by 25% and registers by 1%. We believe that a highly fine-grained virtual dataplane composition approach is likely to increase the consumption of BRAM and register resources. Alternately, designers can choose to embed more features within each component, allowing for tradeoffs between modularity and logic resources. The onion router consumes an additional 5% slices, 3% LUTs and 2% registers beyond the consumption of the IPv4 design example. Table 3 summarizes the detailed logic resource usage and code size for each ReClick component.

4.3 Comparison of ReClick with Other Frameworks

To provide a fair evaluation, we compare the throughput and resource consumption of a ReClick-generated IPv4 dataplane with throughout and resource consumption of IPv4 dataplanes described in SwitchBlade [6] and Chimpp [22]. All evaluated dataplanes were implemented in a Virtex II FPGA available on the NetFPGA 1G platform. The resource utilization of SwitchBlade and Chimpp IPv4 dataplanes were obtained from previously published research data [6] [22]. The IPv4 router described in Chimpp uses 4%

Table 4: **Resource Utilization of ReClick IPv4 and onion routers on a Virtex II Pro**

	NetFPGA IPv4 router	ReClick IPv4 router	ReClick Onion router
Slices	14640	14562	15599
Slice FF	15801	16439	17115
LUTs	23669	23470	24625
IO	356	356	356
BRAMS	25	31	31

more slices than the reference handcoded design. In contrast, the logic utilization of the ReClick router is comparable to the handcoded implementation. The base SwitchBlade platform features dataplane components supporting preprocessor blocks for OpenFlow, IPv6, variable bit extraction and PathSplicing supporting up to four IPv4 dataplanes. This configuration uses approximately 79% of available 4-input LUTs, 89% of available slices and 42% of slice flip flops. The base ReClick IPv4 router features only preprocessing blocks for IPv4 routing and hence consumes 27% fewer slices and 7% fewer registers when compared to the SwitchBlade platform. Since ReClick supports component sharing between dataplanes, we expect the resource usage to grow sublinearly with the number of dataplanes hosted in the virtualization platform. All the dataplanes support line rate forwarding (1 Gbps).

5. CONCLUSION

We have presented ReClick, an extensible modular dataplane design framework for network virtualization. By exposing a simple and intuitive programming model, ReClick enables network developers to quickly adopt reconfigurable hardware for the design and deployment of virtual dataplanes. The ability to integrate custom packet processing blocks, specified in conventional hardware description languages, enhances the flexibility of the framework. The architecture and programming model built into the framework allows logic and memory resources of the reconfigurable hardware platform to be efficiently used without compromising packet forwarding performance. In the future we plan to provide a mechanism for design feedback in the ReClick framework, allowing for quick feedback to developers. Finally, we plan to make ReClick available to the academic research community to encourage the addition of a rich set of features and components for modern packet processing systems.

6. ACKNOWLEDGMENTS

The work was funded in part by National Science Foundation grant CNS-0831940. The FPGA compilation tools were generously donated by Xilinx Corporation.

7. REFERENCES

[1] Cisco Nexus 1000V series switch. http://www.cisco.com/en/US/products/ps9902/.

[2] The Click modular router. http://read.cs.ucla.edu/click.

[3] NetFPGA user's guide. http://yuba.stanford.edu/NetFPGA/static/guide.html.

[4] OpenVZ project page. http://www.openvz.org/.

[5] M. Anwer and N. Feamster. Building a fast, virtualized data plane with programmable hardware. In *Proceedings of the ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, pages 1–8, Aug. 2009.

[6] M. B. Anwer, M. Motiwala, M. B. Tariq, and N. Feamster. SwitchBlade: A platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM*, pages 183–194, Aug. 2010.

[7] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI veritas: Realistic and controlled network experimentation. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 3–14, Sept. 2006.

[8] S. Bhatia, M. Motiwala, W. Muhlbauer, Y. Mundada, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford. Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In *Proceedings of the ACM Conference on Emerging Network Experiment and Technology*, pages 72–77, Dec. 2008.

[9] G. Brebner. Packets everywhere: The great opportunity for field programmable technology. In *IEEE International Conference on Field-Programmable Technology*, pages 1–10, Dec. 2009.

[10] G. Brebner, P. James-Roxby, E. Keller, and C. Kulkarni. Hyper-programmable architectures for adaptable networked systems. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 328–338, Sept. 2004.

[11] N. M. Chowdhury and R. Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862–876, Apr. 2010.

[12] G. A. Covington, G. Gibb, J. W. Lockwood, and N. McKeown. A packet generator on the NetFPGA platform. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 235–238, Apr. 2009.

[13] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, pages 303–320, Aug. 2004.

[14] N. Feamster, L. Gao, and J. Rexford. How to lease the Internet in your spare time. *ACM SIGCOMM Computer Communication Review*, 37(1):1256–1261, Jan. 2007.

[15] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *Proceedings of the ACM Design Automation Conference*, pages 343–348, June 2002.

[16] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting route caching: The world should be flat. In *Proceedings of the International Conference on Passive and Active Network Measurement*, pages 3–12, Apr. 2009.

[17] M. Labrecque, J. G. Steffan, G. Salmon, M. Ghobadi,

and Y. Ganjali. NetThreads: Programming NetFPGA with threaded software. In *Proceedings of the NetFPGA Developer Workshop*, Aug. 2009.

[18] G. Lu, Y. Shi, C. Guo, and Y. Zhang. CAFE: A configurable packet forwarding engine for data center networks. In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow*, pages 25–30, Aug. 2009.

[19] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path splicing. *SIGCOMM Comput. Commun. Rev.*, 38(4):27–38, Aug. 2008.

[20] C. Neely, G. Brebner, and W. Shang. ShapeUp: A high-level design approach to simplify module interconnection on FPGAs. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 141–148, May 2010.

[21] P. Nikander, B. Nyman, T. Rinta-aho, S. D. Sahasrabuddhe, and J. Kempf. Towards software-defined silicon: Experiences in compiling Click to NetFPGA. In *European NetFPGA Developers Workshop*, Sept. 2010.

[22] E. Rubow, R. McGeer, J. Mogul, and A. Vahdat. Chimpp: A Click-based programming and simulation environment for reconfigurable networking hardware. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 36:1–36:10, Oct. 2010.

[23] S. D. Sahasrabuddhe, H. Raja, K. Arya, and M. P. Desai. Ahir: A hardware intermediate representation for hardware generation from high-level programs. In *Proceedings of the IEEE International Conference on VLSI Design*, pages 245–250, Jan. 2007.

[24] J. Turner, P. Crowley, , J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, C. Wiseman, and D. Zar. Supercharging PlanetLab: A high performance, multi-application, overlay network platform. In *Proceedings of the ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 85–96, Aug. 2007.

[25] D. Unnikrishnan, R. Vadlamani, Y. Liao, A. Dwaraki, J. Crenne, L. Gao, and R. Tessier. Scalable network virtualization using FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 219–228, Feb. 2010.

[26] G. Watson, N. McKeown, and M. Casado. NetFPGA: A tool for network research and education. In *Proceedings of the Workshop on Architectural Research Using FPGA*, pages 160–161, Feb. 2006.

[27] D. Yin, D. Unnikrishnan, Y. Liao, L. Gao, and R. Tessier. Customizing virtual networks with partial FPGA reconfiguration. *ACM SIGCOMM Computer Communication Review*, 41(1):125–132, Jan. 2011.

[28] L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron. Cashmere: Resilient anonymous routing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 301–314, Aug. 2005.